Parcial Recuperatorio – 2017s2 - Estructuras de Datos – UNQ

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y tipos de datos vistos en la práctica y en la teórica, salvo que el enunciado indique lo contrario.*
- *No se olvide de poner nombre, nro. de hoja y cantidad total de hojas en cada una de las hojas.*
- ***Deje un ESPACIO VACÍO de 10cm al principio de la 1era hoja del examen.***
  PRESTE ATENCIÓN. ES IMPORTANTE.
- *Le recomendamos leer el enunciado en su totalidad y organizar sus ideas antes de comenzar la resolución. Dedique el tiempo necesario para entender la estructura planteada antes de comenzar a resolver.*
- *Recuerde que la intención es medir cuánto comprende usted del tema. Por ello, no dude en escribir todo lo que usted tiene en cuenta para resolver un ejercicio, en probar sus funciones con ejemplos, etc.*

Nave Espacial Enterprise

La nave espacial *Enterprise* perdió su base de datos y requiere de su ayuda. Su misión es modelar una $Nave$ con información sobre los tripulantes en los distintos sectores de la nave e información sobre el rango (militar) de los tripulantes.

- `naveVacia :: [Sector] -> Nave`
  Propósito: Crea una nave con todos esos sectores sin tripulantes.
  Precondición: la lista de sectores no está vacía
  Costo: $O(S \log S)$ siendo S la cantidad de sectores de la lista.

- `tripulantesDe :: Sector -> Nave -> Set Tripulante`
  Propósito: Obtiene los tripulantes de un sector.
  Costo: $O(\log S)$ siendo S la cantidad de sectores.

- `sectores :: Nave -> [Sector]`
  Propósito: Denota los sectores de la nave
  Costo: $O(S)$ siendo S la cantidad de sectores.

- `conMayorRango :: Nave -> Tripulante`
  Propósito: Denota el tripulante con mayor rango.
  Precondición: la nave no está vacía.
  Costo: $O(1)$.

- `conMasTripulantes :: Nave -> Sector`
  Propósito: Denota el sector de la nave con más tripulantes.
  Costo: $O(1)$.

- `conRango :: Rango -> Nave -> Set Tripulante`
  Propósito: Denota el conjunto de tripulantes con dicho rango.
  Costo: $O(P \log P)$ siendo P la cantidad de tripulantes.

- `sectorDe :: Tripulante -> Nave -> Sector`
  Propósito: Devuelve el sector en el que se encuentra un tripulante.
  Precondición: el tripulante pertenece a la nave.
  Costo: $O(S \log S \log P)$ siendo S la cantidad de sectores y P la cantidad de tripulantes.

- `agregarTripulante :: Tripulante -> Sector -> Nave -> Nave`
  Propósito: Agrega un tripulante a ese sector de la nave.
  Precondición: El sector está en la nave y el tripulante no.
  Costo: No hay datos (justifique su elección).

Los tipos *Tripulante*, *Rango* y *Sector* son abstractos. Pero sabemos que son comparables (*Ord*, *Eq*) y que el tipo *Tripulante* posee esta función en su interfaz (la única que nos interesa):

- `rango :: Tripulante -> Rango`

Ejercicios

Recordatorio: De existir, agregue las precondiciones en las funciones solicitadas. ¡No deje de dividir en subtareas! Y no olvide además incluir propósito y precondiciones de las funciones auxiliares que necesite programar.

a) Implementar las siguientes funciones como usuario del TAD *Nave*, establecer su eficiencia y justificarla:

a) `tripulantes :: Nave -> Set Tripulante`
Propósito: Denota los tripulantes de la nave

b) **Opcional (Bonus):** `bajaDeTripulante :: Tripulante -> Nave -> Nave`
Propósito: Elimina al tripulante de la nave.
Pista: Considere reconstruir la nave sin ese tripulante.

b) Implementar el TAD *Nave* suponiendo el siguiente tipo de representación:

```
data Nave = MkN (Map Sector (Set Tripulante)) (Heap Tripulante) (Sector, Int)
```

Donde:

- Cada tripulante puede estar en un sector como máximo.
- Se guarda al sector con más tripulantes de la nave y cuántos tripulantes tiene ese sector.
- Los tripulantes se ordenan por rango de mayor a menor en la *Heap*
(no se confunda, `findMin` devuelve al tripulante con mayor rango).

a) Escribir los invariantes de representación para poder crear elementos válidos del TAD.

b) Implementar las funciones de la interfaz, calculando eficiencia. Justifique en cada caso la eficiencia obtenida.

Recordatorio:

La interfaz de `Map`, siendo N la cantidad de claves distintas en el map:

`emptyM :: Map k v`	$O(1)$
`assocM :: k -> v -> Map k v -> Map k v`	$O(\log N)$
`lookupM :: Map k v -> k -> Maybe v`	$O(\log N)$
`removeM :: Map k v -> k -> Map k v`	$O(\log N)$
`domM :: Map k v -> [k]`	$O(N)$

La interfaz de `Set`, siendo N la cantidad de elementos del conjunto:

`emptyS :: Set a`	$O(1)$
`addS :: a -> Set a -> Set a`	$O(\log N)$
`removeS :: a -> Set a -> Set a`	$O(\log N)$
`belongs :: a -> Set a -> Bool`	$O(\log N)$
`union :: Set a -> Set a -> Set a`	$O(N \log N)$
`intersection :: Set a -> Set a -> Set a`	$O(N \log N)$
`set2list :: Set a -> [a]`	$O(N)$
`sizeS :: Set a -> Int`	$O(1)$

La interfaz de `Heap`, siendo N la cantidad de elementos de la heap:

`emptyH :: Heap a`	$O(1)$
`isEmptyH :: Heap a -> Bool`	$O(1)$
`insertH :: a -> Heap a -> Heap a`	$O(\log N)$
`findMin :: Heap a -> a`	$O(1)$
`deleteMin :: Heap a -> Heap a`	$O(\log N)$
`splitMin :: Heap a -> (a, Heap a)`	$O(\log N)$